Research Report 230

Automated Timing Analysis of Real-time Programs

W. Ferreira, M.R. Hill, M. Joseph

RR230

ReStat is an interactive software tool for statically analysing the timing properties of real-time programs. Given a distributed program, a description of an architecture and a scheduling discipline. ReStat can be used to determine whether the implementation of the program on the architecture will meet a set of timing deadlines. The program may consist of a number of communicating processes and one process or more may be assigned to any processor. Each process is assumed to have a simple, nonterminating,cyclic structure.
ReStat uses a method of static analysis which establishes the timing properties of a real-time program by examining finite execution segments. The method is syntactic and can be efficiently implemented. This paper gives a outline formal specification of the present version of the tool and illustrates its use as a design aid by way of examples.

Keywords: deadlines, static analysis, scheduling, formal specification

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

December 1992

Automated Timing Analysis of Real-time Programs

W. Ferreira, M.R. Hill, M. Joseph *

Department of Computer Science,
University of Warwick,
Coventry CV4 7AL,
U.K.

November 30, 1992

Abstract

ReStat is an interactive software tool for statically analysing the timing properties of real-time programs. Given a distributed program, a description of an architecture and a scheduling discipline, ReStat can be used to determine whether the implementation of the program on the architecture will meet a set of timing deadlines. The program may consist of a number of communicating processes and one process or more may be assigned to any processor. Each process is assumed to have a simple, nonterminating, cyclic structure.

ReStat uses a method of static analysis which establishes the timing properties of a real-time program by examining finite execution segments. The method is syntactic and can be efficiently implemented. This paper gives a outline formal specification of the present version of the tool and illustrates its use as a design aid by way of examples.

Keywords: deadlines, static analysis, scheduling, formal specification

1 Introduction

A safety-critical program must be proved to meet various requirements under a range of different environmental conditions. Such programs are usually called *reactive* as they are required to produce a response to each stimulus from the environment. They are also time-critical, as a response is required to be produced within a fixed interval of time after the

*Supported by Grant GR/F60021 of the Science and Engineering Research Council

arrival of a stimulus. This paper describes a method and a software tool which can be used to determine the timing properties of a real-time program and to prove whether the implementation of the program will meet its timing requirements.

The timing properties of a program are affected by several factors: the language used in the program, the characteristics of the compiled program, the architecture of the system on which the program is executed and the scheduler used for allocating the resources of the system to the program. One way to estimate these timing properties is by simulation, but for a complex system this will usually require some compromise in the degree of precision used in the simulation if the time taken for the simulation is not to be excessive. Further approximation or estimation will be needed if the system contains infinite (non-terminating) programs, since the simulation can be run only for some chosen length of time. Another approach is to use different analytical methods for different aspects of the problem – such as timing the commands in the program, and characterizing the operating system scheduler – but their results must then be combined to obtain the timing properties of the particular implementation of the program. Yet another solution is to simplify the problem so that it is more easily analysed: for example, by making the program deterministic, restricting the implementation to have just one processor or removing the need for a run-time scheduler by using a pre-determined program schedule.

This paper describes a software tool, ReStat, which has been implemented to address this problem. For a given programming language and a fixed set of processor types, ReStat allows a software designer to map a real-time program to the processors in the architecture and determine whether the mapping is *feasible*, i.e. whether it satisfies all the timing deadlines of the program. The analysis is syntactic, and can be efficiently implemented; it is also conservative, as a deadline will be proved to be satisfied if and only if it is satisfied under all syntactically possible executions of the program.

The class of programs considered includes sequential programs, concurrent programs and distributed programs. A sequential program is considered as a single process program; concurrent and distributed programs may have a fixed number of processes (with no nested parallelism). Programs may have bounded non-determinism in their execution. A process is required to have a simple, cyclic structure with infinite repetition of a sequence of terminating commands. Communication between processes takes place by synchronous or asynchronous message-passing.

The architecture of the system may also be fairly general, ranging from a single processor system to a distributed system with communication links between processors. It is assumed that the timing of machine-level commands is available. Finally, since a number of processes may share a processor for their execution, the scheduling discipline under which processes are executed on each processor must be specified.

ReStat is an interactive tool and makes it possible for a designer to examine alternatives and evaluate the effects of making different choices. A single version of ReStat will deal with a particular programming language and a fixed number of processor types and scheduling disciplines, but the method is sufficiently general for versions to be constructed for many

different requirements.

1.1 Overview

In the following sections, we describe the method of analysis and the general features of ReStat. We use a mixture of formal notation (Z [4]) and text.

Section 2 describes a simple real-time programming language which we shall use to construct parallel programs. We show (Section 3) how to generate *syntactic* executions of such programs by combining syntactic executions of processes, and (Section 4) how one or more process executions can be mapped to each processor in the system. The method of analysis is then described and conditions for the feasibility of an implementation are defined. Finally, we illustrate the operation of ReStat using two examples

2 A Simple Real-Time Language

Consider a simple real-time programming language in which a program consists of one or more sequential processes which communicate with each other through unidirectional channels. Each process is composed from a sequence of simple and structured commands. For this description, we shall refer to but not define: variables, expressions, and statements; we omit procedures and function constructs (the definition can readily be extended to include these features).

Let the sets LABEL, PID, VAR, EXPR, BEXPR and CHAN represent respectively, labels, process identifiers, variables, expressions, boolean expressions and channel names. Then the simple commands of the language can be defined as follows, using the Z free-type notation.

$$\begin{aligned} SCOMMAND ::= & \ \mathbf{skip} \mid \mathbf{delay}\langle\!\langle \mathbb{N} \rangle\!\rangle \mid \mathbf{assign}\langle\!\langle VAR \times EXPR \rangle\!\rangle \\ & \mid \mathbf{input}\langle\!\langle CHAN \times VAR \rangle\!\rangle \\ & \mid \mathbf{output}\langle\!\langle CHAN \times EXPR \rangle\!\rangle \end{aligned}$$

Informally, **skip** is a command that always terminates and takes no time to execute. The command **delay(n)** delays execution of a process for n time units and **assign(v,e)** assigns the value of expression e to variable v. Communication between two processes is through a unidirectional channel and the commands **input(c,v)** and **output (c,e)** represent respectively the input of a value on channel c into variable v and the output of the value of expression e onto channel c.

guard(b) is a constructor for a guarded command which consists of a sequence of alternatives, each with a boolean guard expression and a command. An alternative is 'enabled' if its guard is true; more than one guard may be enabled at the same time.

$$GCOMMAND ::= \mathbf{guard}\langle\!\langle \mathrm{seq}(BEXPR \times COMMAND) \rangle\!\rangle$$

A commands in the language is a labelled simple command (a command in *SCOMMAND*), a labelled guarded command or a sequences of such commands.

$$\begin{aligned} COMMAND ::= & \ \mathbf{atomic}\langle\!\langle LABEL \times SCOMMAND \rangle\!\rangle \\ | & \ \mathbf{if}\langle\!\langle LABEL \times GCOMMAND \rangle\!\rangle \\ | & \ \mathbf{while}\langle\!\langle LABEL \times GCOMMAND \times \mathbb{N} \rangle\!\rangle \\ | & \ \mathbf{seq}\langle\!\langle COMMAND \times COMMAND \rangle\!\rangle \end{aligned}$$

The **if** command permits a choice in the execution of its guarded alternatives. If none of the boolean expressions in the guarded command evaluates to **true** then the execution of **if** is undefined. If at least one guard is true then an alternative corresponding to a true guard is chosen non-deterministically for execution.

The **while** command is repeated as long as any guard is true: on each iteration, an alternative corresponding to a true guard is chosen for execution, as with the **if** command. There is an upper bound to the number of iterations before the command terminates.

A process is made up of a finite sequence of commands. It consists of two parts: an *initialization* part which is executed once, and a *cycle* part which is executed repeatedly forever. Every command has a *position* in either the initialization part or the cycle part of a process.

$$POSITION ::= \mathbf{init} \mid \mathbf{cycle}$$

Each command in a process is uniquely labelled and the function **labels** returns the label of each command.

PROCESS

$$\begin{aligned} & init, cycle : COMMAND \\ & plabels : \mathbb{P}\ LABEL \\ & b : \mathrm{bag}\ LABEL \\ \hline & b = \mathbf{labels}(init) \uplus \mathbf{labels}(cycle) \\ & plabels = \mathrm{dom}(b) \\ & \forall l \in \mathrm{dom}(b) \bullet b(l) = 1 \end{aligned}$$

For convenience, we will sometimes refer to a process with its labels hidden.

$$PROCESS_1 \mathrel{\widehat=} PROCESS \setminus \{b\}$$

A *program* is a fixed set of processes, each labelled with a unique identifier. The commands of each process have a distinct set of labels, and the function *source* maps a label to its process. The function *position* returns the position of any labelled command in the program.

$$
\begin{array}{|l}
\textit{PROGRAM} \\
\hline
\textit{program} : \textit{PID} \rightarrowtail\!\!\!\!\!\!\rightarrow \textit{PROCESS}_1 \\
\textit{source} : \textit{LABEL} \rightarrow \textit{PID} \\
\textit{position} : \textit{LABEL} \rightarrow \textit{POSITION} \\
\hline
\bigcap \{p \mid p \in \mathrm{ran}(\textit{program}) \bullet p.\textit{plabels}\} = \varnothing \\
\forall p : \mathrm{dom}\, \textit{program} \bullet \\
\quad \forall l : \textit{LABEL} \mid \textit{source}(l) = p \Leftrightarrow l \in \textit{program}(p).\textit{plabels} \\
\mathrm{dom}(\textit{source}) = \mathrm{dom}(\textit{position}) \\
\hline
\end{array}
$$

3 Syntactic Executions

From the syntax rules of the previous section we can construct a syntax tree for any legal program. The syntactically possible *executions* of a program are then obtained by traversing its syntax tree. Each traversal produces a finite set of sequences of commands, each sequence being a syntactic process execution made up of the executions of its commands.

There is exactly one syntactically possible execution of each *atomic* command, but there are several syntactically possible executions of **if** and **while** commands, depending on the evaluation of their boolean guards. An *executed* command, or *e-command* is a simple command in SCOMMAND

$$\textit{ECOMMAND} \mathrel{\widehat=} \textit{SCOMMAND} \cup \textit{GCOMMAND}$$

and an execution of a program or process is a sequence of *e-commands* and their corresponding labels.

$$\textit{EXECUTION} == \mathrm{seq}(\textit{LABEL} \times \textit{ECOMMAND})$$

The execution path of each command c is a mapping from *COMMAND* to *ECOMMAND*. Let the execution paths of the **atomic**, **if**, *while* and **seq** commands be respectively

$$[\textit{EPAtomic},\ \textit{EPIf},\ \textit{EPWhile},\ \textit{EPSeq}]$$

where the executions of a **while** command represent the 'unrollings' of its iterative execution.

$$\textit{EPATHS} \mathrel{\widehat=} (\textit{EPAtomic} \land \textit{EPIf} \land \textit{EPWhile} \land \textit{EPSeq})$$

All processes have non-terminating executions as the cycle part of each process never terminates. So there are in general many possible executions of a process, and each is of infinite length. Let seq_ω represent the set of all infinite sequences.

$$\mathrm{seq}_\omega\, X == \{f : \mathbb{N}_1 \rightarrow X\}$$

Let us define the operation of *extension* on sets of sequences so that p^ω defines a set of infinite sequences, each being the concatenation of an infinite number of sequences from the set p. No order is placed on these sequences, and any sequence may be chosen zero or more times.

$$
\begin{array}{|l}
[X] \\
\hline
\_^\omega : \mathbb{P}\,\mathrm{seq}\,X \rightarrow \mathbb{P}\,\mathrm{seq}_\omega\,X \\
\hline
\forall p : \mathbb{P}\,\mathrm{seq}\,X \bullet \\
\quad p^\omega = \{s : \mathrm{seq}_\omega(\mathrm{seq}\,X) \mid \mathrm{ran}\,s \subseteq p \bullet \frown / s\} \\
\hline
\end{array}
$$

The syntactic executions of a process is a set of (infinite) sequences, each of which consists of a finite sequence of commands from the initialization part of the process concatenated with any sequence from the unrolling of the cycle part.

$$
\begin{array}{|l}
\mathit{SYExecutions} \\
\hline
\Xi \mathit{PROGRAM} \\
\mathit{EPATHS} \\
\mathit{syexe} : \mathit{PID} \rightarrow \mathbb{P}(\mathrm{seq}_\omega(\mathit{LABEL} \times \mathit{ECOMMAND})) \\
\hline
\forall p \in \mathrm{dom}\,\mathit{program} \bullet \\
\quad \mathit{syexe}(p) = \\
\qquad \{s \in \mathit{epaths}(\mathit{program}(p).\mathit{init}), \\
\qquad\qquad t \in (\mathit{epaths}(\mathit{program}(p).\mathit{cycle}))^\omega \bullet s \frown t\} \\
\hline
\end{array}
$$

4 Processor Executions

Each process in the program must be allocated to a processor for its execution. There are many ways in which this allocation can be done: all the processes may be placed on one processor, or each process may be assigned to a separate processor (giving a *maximum parallel* allocation) or some allocation may be chosen between these limits. If two or more processes are assigned to a processor, then the processes must share the processor according to a scheduling policy. Given the individual executions of each process, a *processor execution* is an interleaving of one execution of each process, according to a scheduling policy. There will be a set of such executions for each processor and it will be seen that the processor executions of a program are a subset of its syntactic executions.

Let *PRD* be the set of all processor identifiers. Each processor can be executing any number of processes and each process in the set *PID* of processes is allocated to one processor.

$$
\begin{array}{|l}
\mathit{MAPPING} \\
\hline
\mathit{alloc} : \mathit{PRD} \leftrightarrow \mathit{PID} \\
\hline
\forall p \in \mathrm{ran}\,\mathit{alloc} \bullet \#\mathit{alloc}^\sim(\!|\{p\}|\!) = 1 \\
\hline
\end{array}
$$

Each element in the sequence of commands executed by a processor belongs to one of the processes allocated to that processor. As there are many possible executions of each process, we can define a *processor schedule*, *PSCHEDULE*, to contain the *merged* sequences obtained from one execution each of these processes; if there is just one process on a processor, then the processor schedule is the same as the executions of the process.

$$
\begin{array}{|l}
\textit{PSCHEDULE} \\
\hline
\textit{merged} : \mathrm{seq}_{\omega}(\textit{LABEL} \times \textit{ECOMMAND}) \\
\textit{traces} : \textit{PID} \pfun \mathrm{seq}_{\omega}(\textit{LABEL} \times \textit{ECOMMAND}) \\
\hline
\end{array}
$$

Each process execution is infinite because there are repeated executions of its **cycle** command. We shall later need to distinguish between different iterations of a **cycle** command. Let the function *cycleno* return the repetition number of the cycle in which a labelled command is executed; e.g. $\textit{cycleno}(l, i) = j$ represents the execution of command l at position i in the jth cycle in process p, where $\textit{source}(l) = p$.

A *program schedule*, *SCHEDULE*, is a collection of processor schedules, one for each processor.

$$
\begin{array}{|l}
\textit{SCHEDULE} \\
\hline
\textit{MAPPING} \\
\textit{pschedules} : \textit{PRD} \pfun \textit{PSCHEDULE} \\
\textit{cycleno} : (\textit{LABEL} \times \mathbb{N}) \pfun \mathbb{N} \\
\hline
\forall p \in \mathrm{dom}(\textit{alloc}) \bullet \\
\quad \textit{alloc}(\!|\{p\}|\!) = \mathrm{dom}(\textit{pschedules}(p).\textit{traces}) \\
\hline
\end{array}
$$

Not all the syntactic executions generated will be valid executions of the program as some merged executions will contain mismatched communications between processor schedules, i.e. there will be more inputs than outputs on some channel, or vice-versa. We assume that programs are free of deadlock, so we need to consider only those merged executions for which all communications between processes are matched.

Let the relation comp define the set of all complementary pairs of communications.

$$
\begin{array}{|l}
\mathsf{comp} : (\textit{PID} \times \textit{ECOMMAND}) \rel (\textit{PID} \times \textit{ECOMMAND}) \\
\hline
\forall p_1, p_2 : \textit{PID};\ c_1, c_2 : \textit{ECOMMAND};\ v : \textit{VAR};\ e : \textit{EXPR} \bullet \\
\quad (p_1, \mathbf{input}(p_2, v))\ \mathsf{comp}\ (p_2, \mathbf{output}(p_1, e)) \\
\quad (p_1, \mathbf{output}(p_2, e))\ \mathsf{comp}\ (p_2, \mathbf{input}(p_1, v)) \\
\end{array}
$$

The schema *MATCH* introduces a relation cmatch which identifies matching pairs of communication commands from processes. Let the function trunc truncate an infinite sequence and return a specified finite sequence. Then a command (lab_1, c_1) from a processor schedule ps_1 matches a command (lab_2, c_2) from a processor schedule ps_2 if they are complementary

communication commands (i.e. in the relation comp), and the number ord_1 of inputs received equals the number ord_2 of outputs.

$$
\begin{array}{|l}
\textit{MATCH} \\
\hline
\textit{MAPPING} \\
\textit{PROGRAM} \\
\mathsf{cmatch} : (PRD \times \mathbb{N}) \leftrightarrow (PRD \times \mathbb{N}) \\
c_1, c_2 : ECOMMAND \\
ord_1, ord_2, ind_1, ind_2 : \mathbb{N} \\
p_1, p_2 : PRD \\
ps_1, ps_2 : PSCHEDULE \\
lab_1, lab_2 : LABEL \\
pschedules : PRD \pfun PSCHEDULE \\
\hline
\left(\begin{array}{c}
pschedules(p_1) = ps_1 \\
pschedules(p_2) = ps_2 \\
c_1 = second(ps_1.merged(ind_1)) \\
c_2 = second(ps_2.merged(ind_2)) \\
lab_1 = first(ps_1.merged(ind_1)) \\
lab_2 = first(ps_2.merged(ind_2)) \\
ord_1 = \#((map\ second(\mathsf{trunc}(ps_1.merged, ind_1))) \rhd \{c_1\}) \\
ord_2 = \#((map\ second(\mathsf{trunc}(ps_2.merged, ind_2))) \rhd \{c_2\}) \\
ord_1 = ord_2 \\
\mathsf{comp}((source(lab_1), c_1), (source(lab_2), c_2))
\end{array} \right) \\
\Rightarrow \\
\mathsf{cmatch}(p_1, ind_1) = (source(lab_2), ind_2)
\end{array}
$$

$$MATCH_1 \mathrel{\widehat=} MATCH \setminus \{c_1, c_2, ord_1, ord_2, p_1, p_2, ps_1, ps_2, ind_1, ind_2, lab_1, lab_2\}$$

4.1 Valid Executions

Let the condition *ValidExecution* be true if all pairs of commands from one process in any schedule p have the same order that they have in some execution of the process, i.e. the execution order of every two such commands is preserved. In addition, let the schema *ValidMerge* define the property that the merged sequence of each processor schedule is a valid merged sequence of the process traces on the given processor and follows the processor scheduling policy (note that the scheduling policy is processor dependent, and the the merge must be specified for each type of processor). Then a program schedule $SCHEDULE_1$ on a set of processors can be defined as

$$SCHEDULE_1 \mathrel{\widehat=} ValidExecution \land ValidMerge$$

Processor executions can be defined in terms of the syntactic executions of Section 2. Each processor execution is a valid schedule whose commands are drawn from some syntactic

execution of the program. The merged sequences defined by these commands correspond to the scheduling policy defined by ValidMerge.

PREexecutions

$$
\begin{array}{l}
SYExecutions \\
schedules : \mathbb{P}\ SCHEDULE_1 \\
\hline
\forall s \in schedules \bullet \\
\quad \forall ps \in \mathrm{ran}(s.pschedules) \bullet \\
\quad\quad \forall p \in \mathrm{dom}(ps.traces) \bullet \\
\quad\quad\quad ps.traces(p) \in syexe(p)
\end{array}
$$

4.2 Deadlines

Commands in a program may be annotated with *deadlines*. A deadline expresses a timing constraint between the communication commands of synchronised processes, and has the form "if a command C starts (or finishes) execution at time t then in the time interval $(t, t+d], d > 0$, one of the commands in the deadline set $\{C_1 .. C_n\}$ must start or finish execution" [3]. The command C will sometimes be referred to as the deadline command and the interval $(t, t+d], d > 0$ as the deadline interval. For each deadline, a choice must be made of where the deadline interval begins (at the start or finish of C) and ends (at the start or finish of a command in $\{C_1 .. C_n\}$.

$$POINT ::= \mathbf{start} \mid \mathbf{finish}$$

$$DEADLINE ::= \mathbf{dline}\langle\!\langle \mathbb{F}\ LABEL \times POINT \times POINT \times \mathbb{N} \rangle\!\rangle$$

Not all the commands of a program have deadlines so annotations are defined as a partial mapping from LABEL to DEADLINE.

ANNOTATIONS

$$
\begin{array}{l}
an : LABEL \pfun DEADLINE
\end{array}
$$

5 Snapshots and Repetition Points

Having generated a set of valid executions, we must now find a way to analyse these executions and determine their timing properties. To prove that a program does not violate its deadlines it must be proved that the deadlines are not violated in *any* execution of the program. Note that the executions are infinitely long, as processes are non-terminating.

To simplify the analysis of these infinite executions, we can make use of the cyclic nature of each process and the repeatability that this enforces on the behaviour of the program.

Informally, every execution of a process will contain repetitions of its execution history and so the 'execution state' of the program, at some point in an execution, will be repeated at a later point. This will be true for each possible executions of the program. We shall characterise the execution state of a program by a *snapshot* and the point at which this occurs again as the *repetition point.*

Consider for any execution of the program the interval between a snapshot ans its repetition. If any commands in this interval have deadlines that lie beyond the repetition point, let the interval be extended to a *truncation point* which includes the end point of all such deadlines. The method of analysis makes use of these extended intervals to determine the timing properties of the program and based on some results from [3]. Given an extended interval, it was shown there that (a) if a deadline is violated anywhere in an (infinite) execution of a program then it will be violated in such an interval of this execution, and (b) if no deadlines are violated in such an interval of some execution then they will not be violated in the infinite execution which includes that interval.

The schema *START* defines two functions, *start* and *end* which return the start and end times respectively of any indexed command in any merged sequence; *start* may easily be defined inductively on a schedule, depending on the position of the command and its type. If a command is not interrupted, its end time is its start time plus its execution time.

START

$$
\begin{array}{l}
SCHEDULE_1 \\
start : (LABEL \times \mathbb{N}) \rightarrow \mathbb{N} \\
end : (LABEL \times \mathbb{N}) \rightarrow \mathbb{N} \\
cost : LABEL \rightarrow \mathbb{N}
\end{array}
$$

A snapshot taken at some point t in time during the execution of the program contains a description of the activities of all the processes in the program at that point. Some commands may have completed execution at time t but others may be executing; (l, i) describes a command labelled l with i units of time left to execute.

SNAPSHOT

$$
\begin{array}{l}
t : \mathbb{N} \\
s : LABEL \rightarrow \mathbb{N}
\end{array}
$$

A snapshot is defined in terms of two functions, s and u. The function s (for sliced) maps all those labelled commands whose executions are not complete at time t, to the time needed for completion. The function u (for unsliced) maps the next command to begin execution at time t to its execution time. Thus a snapshot represents each process either by the label of its command which has been sliced by t, or by the label of the next complete command to be executed. The schema *TakeSnapshot* makes use of a property *in* to determine if a number lies in an open interval.

$$\_\text{in}\_ : \mathbb{N} \leftrightarrow (\mathbb{N} \times \mathbb{N})$$

$$\forall i : \mathbb{N};\ (x, y) : (\mathbb{N} \times \mathbb{N}) \bullet$$
$$i \text{ in } (x, y) \Leftrightarrow x < i \land i < y$$

TakeSnapshot

$$MAPPING$$
$$PROGRAM$$
$$SCHEDULE_1$$
$$START$$
$$sn : SNAPSHOT$$
$$s, u : LABEL \rightarrow \mathbb{N}$$

$$sn.s = s \cup u$$

$$\forall p \in \text{dom}(program) \bullet$$
$$\exists l : LABEL;\ i : \mathbb{N};\ pr : PRD \bullet$$
$$source(l) = p \land p \in alloc(\!|\{pr\}|\!)$$
$$sn.t \text{ in } (start(l, i), end(l, i))$$
$$l = first(pschedules(pr).merged(i))$$
$$\Rightarrow$$
$$s(l) = (end(l, i) - sn.t)$$

$$\forall p \in \text{dom}(program) \setminus source(\text{dom}(s)) \bullet$$
$$\exists l : LABEL;\ i : \mathbb{N};\ pr : PRD \bullet$$
$$source(l) = p \land p \in \text{dom } alloc(\!|\{pr\}|\!)$$
$$l = first(pschedules(pr).merged(i))$$
$$sn.t \leq start(l, i)$$
$$\forall m : LABEL, j : \mathbb{N} \bullet$$
$$source(m) = p \land m = first(pschedules(pr).merged(j))$$
$$sn.t \leq start(l, j)$$
$$\Rightarrow$$
$$start(l, j) > start(l, i)$$
$$\Rightarrow$$
$$u(l) = cost(second(pschedules(pr).merged(i)))$$

$$TakeSnapshot_1 \mathrel{\widehat=} TakeSnapshot \setminus \{s, u\}$$

Given two snapshots of a program, the schema *RepetitionPoint* defines the conditions under which one snapshot is a repetition of the other. In general, a repeatable snapshot will occur after the initialisation parts of all processes have been completed; the repetition of this snapshot at some later point of time will occur when the execution of each process has advanced by at least one cycle.

RepetitionPoint

$$
\begin{array}{l}
PROGRAM \\
SCHEDULE_1 \\
START \\
ss, srp : SNAPSHOT \\
\hline
ss.t \geq max\{pr \in \mathrm{dom}(pschedules), i \in \mathrm{dom}(pschedules(pr).merged) \mid \\
\quad position(first(pschedules(pr).merged(i))) = \mathbf{init} \bullet end(pr, i)\} \\
ss.t < srp.t \\
\forall l \in \mathrm{dom}(ss.s) \bullet \\
\quad cycleno(srp.s(l)) > cycleno(ss.s(l)) \\
ss.s = srp.s
\end{array}
$$

$$Snapshot \mathrel{\widehat=} TakeSnapshot_1[ss/sn] \land TakeSnapshot_1[srp/sn] \land RepetitionPoint$$

The *truncation point* is the latest end point of a deadline in the execution interval between the snapshot and its repetition. It is the point to which a processor execution must be expanded to define the extended interval. Let the function dtime be a function that returns the time from the freetype *DEADLINE*.

TruncationPoint

$$
\begin{array}{l}
ANNOTATIONS \\
START \\
Snapshot \\
tp, ld : \mathbb{N} \\
\hline
\mathrm{dom}(an) \subseteq \mathrm{dom}(source) \\
\mathrm{dom}(an) = \varnothing \Rightarrow ld = 0 \\
\mathrm{dom}(an) \neq \varnothing \Rightarrow \\
\quad ld = max(\{p \in \mathrm{ran}\, pschedules, s \in \mathrm{ran}(p.traces), (l, c) \in \mathrm{ran}(s) \mid \\
\qquad l \in \mathrm{dom}(an) \bullet cost(c) + \mathsf{dtime}(an(l))\}) \\
tp = srp.t + ld
\end{array}
$$

$$TruncationPoint_1 \mathrel{\widehat=} TruncationPoint \setminus \{ld\}$$

6 Program Feasibility

The implementation of a program is said to be *feasible* if it meets all its deadlines. This feasibility must be established in the extended interval from a snapshot to the truncation

point following the repetition of the snapshot, for every possible schedule of the program. The schema *Feasible* defines the feasibility of a *SCHEDULE*.

If there are no deadlines, then an implementation is trivially feasible. Otherwise, for any command with a deadline, at least one of the labelled commands in the deadline set for that command must meet the following requirements: its execution ends at or before the truncation point and it satisfies the associated deadline. Let point_1, and point_2 be functions that take values in $\{\mathbf{start}, \mathbf{finish}\}$ for a deadline of $\mathsf{dtime}(\mathsf{an}(\mathsf{l}))$ between a deadline command l and a deadline set $\mathsf{dcommands}(an(l))$.

Feasible

$SCHEDULE_1$
$TruncationPoint$
$FeasibleScheds : \mathbb{P}\ SCHEDULE_1$

$$\forall s : SCHEDULE_1 \mid s \in FeasibleScheds \Leftrightarrow$$

$$\mathrm{dom}(an) = \varnothing \vee$$

$$\forall p : PRD;\ i : \mathbb{N};\ (l, c) : (LABEL \times ECOMMAND) \mid$$
$$pschedules(p).merged(i) = (l, c) \wedge$$
$$l \in \mathrm{dom}(an) \wedge start(p, i) \leq srp.t \bullet$$

$$\exists p' : PRD;\ i' : \mathbb{N};\ (l', c') : (LABEL \times ECOMMAND) \mid$$
$$l' \in \mathsf{dcommands}(an(l)) \wedge$$
$$pschedules(p').merged(i') = (l', c') \wedge$$
$$end(p', i') \leq tp$$
$$start(p, i) < start(p', i') \bullet$$

$$(\mathsf{point}_1(an(l)) = \mathbf{start} \wedge \mathsf{point}_2(an(l)) = \mathbf{start} \Rightarrow start(p', i') - start(p, i) \leq \mathsf{dtime}(\mathsf{an}(\mathsf{l})))$$

$$(\mathsf{point}_1(an(l)) = \mathbf{start} \wedge \mathsf{point}_2(an(l)) = \mathbf{finish} \Rightarrow end(p', i') - start(p, i) \leq \mathsf{dtime}(an(l)))$$

$$(\mathsf{point}_1(an(l)) = \mathbf{finish} \wedge \mathsf{point}_2(an(l)) = \mathbf{start} \Rightarrow start(p', i') - end(p, i) \leq \mathsf{dtime}(an(l)))$$

$$(\mathsf{point}_1(an(l)) = \mathbf{finish} \wedge \mathsf{point}_2(an(l)) = \mathbf{finish} \Rightarrow end(p', i') - end(p.i) \leq \mathsf{dtime}(an(l)))$$

If a method of determining feasibility is to be practical, it must also be efficient. In this method, this would mean that the executions are expanded as little as possible, as this will then save storage and the time needed to generate and analyse each execution. The use of

snapshots makes it possible for the analysis to be very efficient, as we illustrate with the following example.

Consider a program P consisting of two processes, p_1 and p_2. Let each process p_i, $i \in \{1,2\}$, consist of a single command E_i which is to be executed once every T_i units of time, and let $T_1 < T_2$. Let the execution time of the command for each process be τ_1 and τ_2 respectively. Assume that the processes do not communicate with each other. To determine the feasibility of executing the processes on the same procesor we must first define a scheduling discipline.

A simple and well-known discipline is rate-monotonic scheduling [2]. This assumes that processes are given priorities depending in inverse order to their repetition periods (i.e. the most frequently repeated process has the highest priority), and that a lower priority process is pre-empted when a higher priority process is ready to begin execution. In this example, since $T_1 < T_2$, p_1 must have higher priority than p_2 and be allowed to pre-empt it.

The maximum processing load on the system occurs when each process starts execution at the same time, e.g. at time 0 [2]. One way to check for the feasibility of the implementation, i.e. to determine whether each process P_i is indeed executed once every T_i units of time, is to determine whether there is a point of time t in the interval $[0, T_2]$, when all demands for computation upto t are satisfied [1]. (Note that this applies in this case because the processes do not communicate with each other).

The feasibility can also be determined using snapshots, and in fact this method has the advantage that it allows for both independent processes and processes that communicate with each other.

In the program P, the requirement for periodically executing each process can be defined by setting a deadline for each process equal to its period. For example, let $T_1 = 5$ and $T_2 = 7$, so that the deadlines are

$D(E_1) = \langle \{E_1\}, 5, start, start \rangle$

$D(E_2) = \langle \{E_2\}, 7, start, start \rangle$

Assume further that $\tau_1 = 1$ and $\tau_2 = 2$. Then the executions of each process can be represented by a timing diagram, as in Fig. 1.

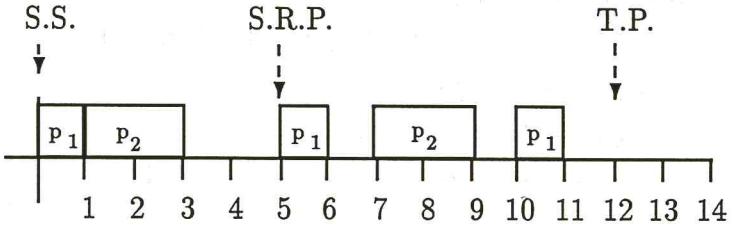


Figure 1 : Feasible execution

The worst case load of the system will occur at time 0 and again at the least common multiple of the process periods, in this case at time 35. If the program is feasible upto this point then

it will always be feasible. But using the method of [1] the feasibility can be established at time 3, at which point there are no unfulfilled requests for computation.

The length of the execution that is required to be examined by the snapshot method is a little longer, as can be seen from Fig. 1. There is no initialisation part in either of the processes and the snapshot (SS) at time 0 is repeated at time 5 (SRP). The truncation point (TP), obtained by extending the deadline interval to include the latest deadline in the interval $[SS, SRP]$, is $5 + 7 = 12$. Thus the implementation is feasible according to this method if all the deadlines in the interval $[0, 12]$ are satisfied.

To show that the snapshot method will also detect infeasiblity, let the computation time of C_2 can be increased to 7 time units. The timing diagram with this change is shown in Fig. 2.

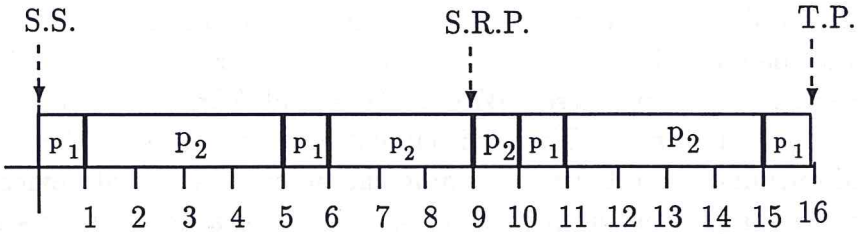


Figure 2 : Infeasible execution

Here it can be seen that the time between successive starts of p_2 is 8 time units, thus failing the required deadline. The infeasibility is detected much before the point of the least common multiple of the process periods.

7 Analysing Programs

The snapshot method of analysis is independent of the language and the system architecture. But naturally any implementation of the method must consider a specific language and processor type. The current implementation of ReStat analyses programs written in Occam^R, a subset of Occam, for execution on a Transputer system.

The Transputer scheduler has two levels of priority with high priority processes able to pre-empt low priority processes (within one level, ready processes are scheduled in round-robin fashion). Pre-emption can only occur when a low priority process is executing one of a set of interruptible commands.

When a real-time program is executed it must communicate with its environment, for example, reading a value from a sensor or sending data to an actuator. The timing characteristics of these devices are important because they affect the timing of the real-time program (in general, the reverse is not true). In ReStat, devices are emulated as 'environment processes'. These processes have a very simple structure (typically communication with a timer and with a 'device process') and may either send values to or receive values from the program;

they may be periodic, aperiodic or sporadic. During the analysis, environment processes are not merged with any other process and are assumed to execute on their own 'processors'.

We shall illustrate the operation of ReStat using as an example a simple embedded real-time control system and consider how the placement of processes on processors can affect their ability to meet deadlines. All the timings are specific to a Transputer architecture and processes are analysed as if executing under the micro-coded Transputer scheduler.

7.1 Embedded Control Example

ReStat can also be used to analyse non-deterministic programs of the form often used in embedded real-time control systems.

Consider a simple cruise control program which monitors the position of the throttle and observes the road speed in order to automatically control the speed of a car. The program has three environment processes *Wheel*, *RThr* and *ThrSink*, and three system processes *RSpeed*, *CtrSp* and *ThrCtr*. The environment processes provide the control processes with inputs and outputs which have the timing characteristics of real devices. For example, the control process *RSpeed* is assumed to be interfaced to a device that sends the speed every second, and this is simulated by process *Wheel*. The environment process *RThr* reads the position of the throttle and the process *ThrSink* sets the position of the throttle.

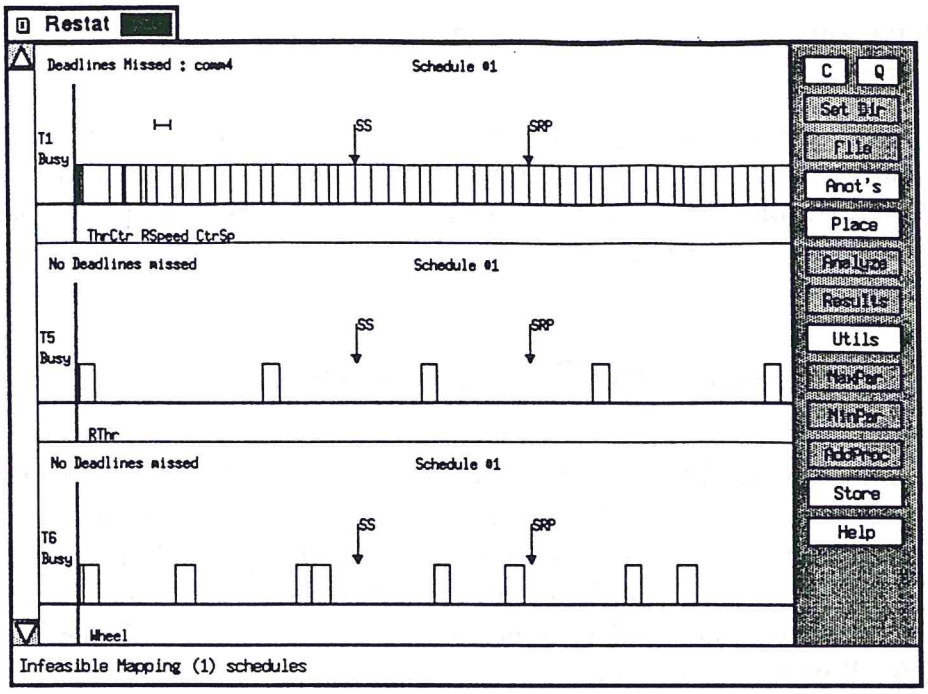
The three control processes *RSpeed*, *CtrSpd* and *ThrCtr* communicate with the environment and with each other to compute any throttle changes. The main timing requirement of the program is to provide a new throttle position within a time T of receiving a new speed reading from the wheel. This can be expressed as a deadline:

$$SpeedIn(\{ThrotOut\}, start, end, T)$$

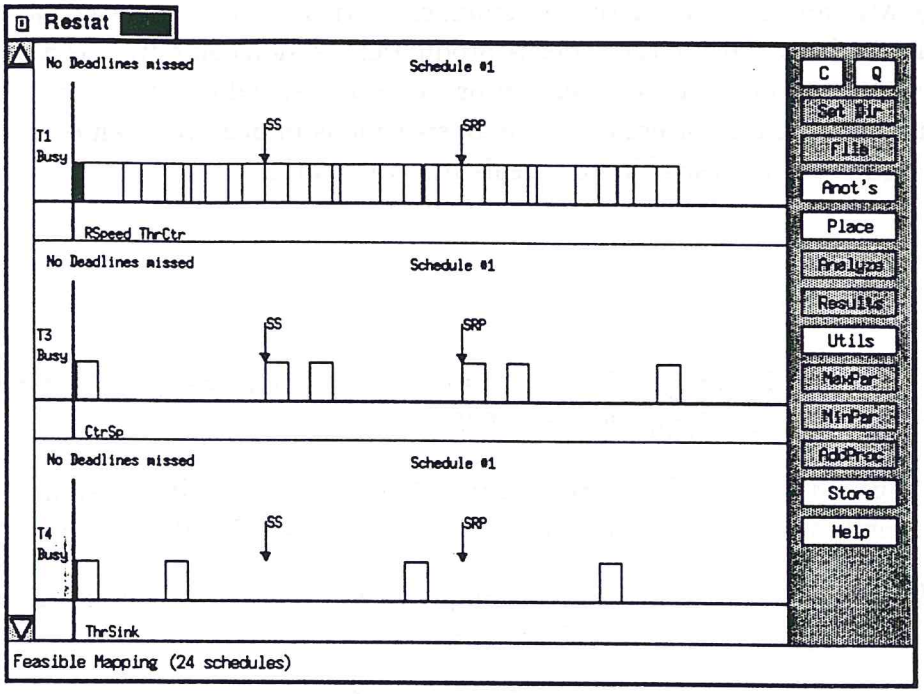
where *SpeedIn* and *ThrotOut* are labels for the appropriate communication commands in the program.

ReStat was used to analyse the execution of this program on different Transputer architectures. The first architecture investigated was the maximum parallel case, i.e. each process executed on a T800 Transputer running at 20Mhz; this was found to be feasible and so an attempt was made to reduce the number of processors. At this stage all the processes were assigned to high priority.

The next attempt, with all three control processes on one T800 Transputer, was found to be infeasible (Fig. 4a shows a timing diagram for each processor and highlights the missed deadline). Another processor was added and the process placement altered so that *ThrCtr* and *RSpeed* shared one T800 whilst *CtrSpd* was executed alone. This allocation was feasible (Fig. 4b).



(a)



(b)

Figure4 : ReStat Results

The placement of processes on is an important factor in the feasibility of a system as timing deadlines are often missed because of delays introduced due to scheduling. ReStat allows the designer to alter other aspects of a system, such as process priority, processor type and speed. For example, by changing the priority of *ThrCtr* in the previous analysis the implementation

becomes infeasible because the execution of *ThrCtr* is delayed by interruptions from the high priority process.

But by changing a processor type or speed, for example by changing the processor executing *ThrCtr* and *RSpeed* to a T212, a previously feasible system can also be made infeasible. This is because though the processor speed is unchanged, floating point operations are much faster on a T800 due to its co-processor.

8 Discussion and Conclusions

ReStat can be used in different ways: to examine the feasibility of different ways of assigning processes to processors, of different choices of processor and of different assignments of priority to each process. Thus, it can serve as a design tool, enabling the system engineer to experiment with the available choices in implementing a real-time program. It can also be used as a validation tool, to determine whether a particular implementation of a program will satisfy all its timing deadlines.

The present version of ReStat has a high-level implementation written in the functional language ML, and this has close correspondence with the formal specification in Z, enabling relatively quick checking of the code. A production version could well be written in a language with lower execution overheads but, in practice, the speed of execution has not really been a limiting factor. More often, a long analysis time is indicative of an over-complex program structure which is undesirable for a real-time application.

References

[1] M Joseph and P Pandya. Finding response times in a real-time system. *The Computer Journal*, Vol.29(No.5):pp390–395, 1986.

[2] C L Liu and J W Layland. Scheduling algorithms for multiprogramming in a hard real time environment. *ACM Journal*, Vol.20(No.1):pp46–61, 1973.

[3] A Moitra and M Joseph. Determining timing properties of infinite real-time programs. Technical report, Dept. CS, Warwick University, 1991.

[4] J M Spivey. *The Z Notation : A Reference Manual.* Prentice-Hall, 1989.